Enabling the business-based
Internet of Things and Services

(FP7 257852)

D9.2.2 Annual Integration and Quality Assurance Report 2

Published by the ebbits Consortium

Dissemination Level: Public

Project co-funded by the European Commission within the 7th Framework Programme
Objective ICT-2009.1.3: Internet of Things and Enterprise environments

Document control page

Document file: D9.2.2 Annual Integration and Quality Assurance Report 2 v.1.0.docx
Document version: 1.0
Document owner: Karl Catewicz (FIT)

Work package: WP9 – Platform integration and deployment
Task: T9.2 – Quality Management, develop monitoring of source code quality and usage of metrics
Deliverable type: R

Document status: ☒ approved by the document owner for internal review
☒ approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.01	Alexander Schneider	2012-05-22	First version with TOC
0.03	Karl Catewicz	2012-08-01	Section 4
0.04	Pietro A. Cultrona	2012-08-09	Added contribution to section 5
0.05	Michael Jacobsen	2012-08-24	Small section on traceability components added
0.06	Karl Catewicz	2012-08-24	Executive summary added
0.07	Mauricio Caceres (ISMB)	2012-08-24	Some lessons learned added
0.08	Peeter Kool	2012-08-27	Test beds
0.09	Karl Catewicz	2012-08-27	Update,lessons learned
0.10	Karl Catewicz	2012-08-29	Conclusions section
0.11	Karl Catewicz	2012-08-30	Reviewers suggestions included
1.0	Alexander Schneider	2012-08-30	Final version submitted to the European Commission

Internal review history:

Reviewed by	Date	Summary of comments
Peter Kostelnik	2012-08-15	Approved with minor comments
Claudio Pastrone	2012-08-15	Approved with minor comments

Index:

1. Executive summary

The “Annual Integration and Quality Assurance Report” deliverable repeatedly reports on the progress made in integration and quality assurance in ebbits from a technical perspective. The report is connected with subtask T9.2 with the aim of improving maintainability and overall quality of software developed in ebbits. The consortium agreed to use seasoned tools and techniques like iterative requirements engineering according to the Volere schema, software configuration management, automated build processes, unit testing, integration testing, web service testing, validation, continuous integration, coding rules, software metrics and code reviews.

The current report follows the path initialized in the D9.2.1. Source code monitoring methods discussed and established in previous deliverable are consequently used in D9.2.2
The theoretical integration and quality concepts are not subject of this document and can be examined in D9.2.1

The current document focuses mostly on the status of current code base (sec. 4) and high level architecture (sec. 5)

Section 4 reports on the current status of quality and integration in ebbits. Main part of it covers tests done with help of Sonar. Significant improvement of the total java based code base was measured in almost all fields like rule compliance, API documentation and code duplications. Code complexity slightly increased and should be tackled in the next iteration. The potential C# code base improvements could not be highlighted by comparison with the previous deliverable version, because D9.2.1 didn’t provide C# data. The total “health” of C# code base is good. It is characterized by very low duplication- and complexity rates. Static rule compliance is quite good too. The API comments rates are below average level.

Section 5 presents an overview of the integration status in ebbits. It describes modules, sub-systems and systems of ebbits.

Section 6 collects lessons learned with respect to integration and quality assurance.

Section 7 concludes this report.

2. Introduction

2.1 Purpose, context and scope of this deliverable

The “Annual Integration and Quality Assurance Report” deliverable repeatedly reports on the progresses made in integration and quality assurance in ebbits from a technical perspective. The report is aimed at sub-tasks of tasks T9.2 (“Quality Management, develop monitoring of source code quality and usage of metrics”) in order to ensure a lasting maintainability and fitness of the software developed in ebbits. With respect to this target, especially internal quality attributes of the code (like Maintainability and Analysability) grow in importance. This task strives to strengthen the understanding of such quality attributes and, consequently, to improve the code quality.

Since the beginning of the project a considerable amount of efforts has been invested in planning and preparing integration and quality assurance. It was a common agreement to adhere to well-known software engineering methodologies to assure an acceptable quality of the software produced by the ebbits consortium. The consortium agreed to use seasoned tools and techniques like iterative requirements engineering according to the Volere schema, software configuration management, automated build processes, unit testing, integration testing, web service testing, validation, continuous integration, coding rules, software metrics and code reviews. The report aims to review the situation regarding adaptation of these tools and processes by the consortium.

3. Report of the Current Situation of Quality in ebbits

3.1 Introduction

This section reports on the current situation of integration and quality in ebbits. It reports what the presumably most critical quality and integration problems in ebbits are (see Section Fehler! Verweisquelle konnte nicht gefunden werden.). Furthermore, this section is updated regularly with current information obtained through quality assurance tools like Java Sonar.

The second version of this report relies on information from Sonar only (see Section Fehler! Verweisquelle konnte nicht gefunden werden.) but later versions will incorporate information from other sources, too.

3.2 Unit and integration testing

Unit- and integration tests are an essential part of modern software development. A short description of both methods follows.

Unit testing is a method to test application units with the help of so the called test cases.
Units are the smallest testable parts of a bigger context.
Units are usually smaller pieces of code like functions, procedures, classes etc.
The idea behind unit testing is to prove that certain piece of code fulfils expectations of code writers or white box testers.
Available unit tests allow the programmer to check if his code is still correct, even after actual implementation was changed.

Unit tests are integral part of Continuous integration (CI) environments. They also help during later integration testing phase, although they are not meant to be a integration substitute.
An example of a unit test in Java (Junit framework):

```
public class testHelloWorld{
		@Test
		public void testHelloWorld() {
			String expected = "HelloWorld";
			String actual = "HelloWorld";
			assertEquals(expected, actual);
		}
}
```

The shown example compares two values, namely the actual and the expected value. If the values are not equal, the unit test called “testHelloWorld” fails. The current example always passes because the values are equal.

Integration testing is a phase of software testing where several modules are combined and tested together. The integration always occurs after unit testing.

The main idea behind integration testing is to verify the function, reliability and performance in an actual environment. While unit tests cover the smallest parts of code, integration tests involve the whole design. A typical integration test interconnects different modules. When such test case is started, it tests the basic functionality of a whole design. D9.1 describes also testing of interfaces as a part of integration process.

An example of such integration test could be a complex webservice. Once all submodules are unit tested, the integration test follows. This requires the actual setup of the running environment, the startup of the environment (container) and the deployment inside it. After the webservice is deployed integration tests can be applied.

External infrastructure, webservices etc. can be mocked to keep the complexity at a manageable level.

3.3 Most critical current quality and integration problems

Source code used to create current report is located in two repositories (see D.9.1). New developed ebbits specific modules are located in https://forge.fit.fraunhofer.de/svn/ebbits . More general LinkSmart middleware can be found here https://linksmart.svn.sourceforge.net/svnroot/linksmart .
Regarding D9.1 this report is using code from trunk of the ebbits repository.

Both source trunks, java and c# lack unit- and integration tests. Even if personal tests exist somewhere, they are not located in an exposed place inside the Svn repository.

Unit- and integration tests are the foundation of modern software development.
The key points of the motivation behind unit testing usage can be found in previous section.
The main reason for both the techniques are the postponed reactions to bugs and integrations problems.
Without any tests, bugs and integration problems sum up at end phases of a software development cycle. Therefore costs of integration of a medium- or big size project can become more time and money consuming.

The following image shows growing repair costs of software in different phases.

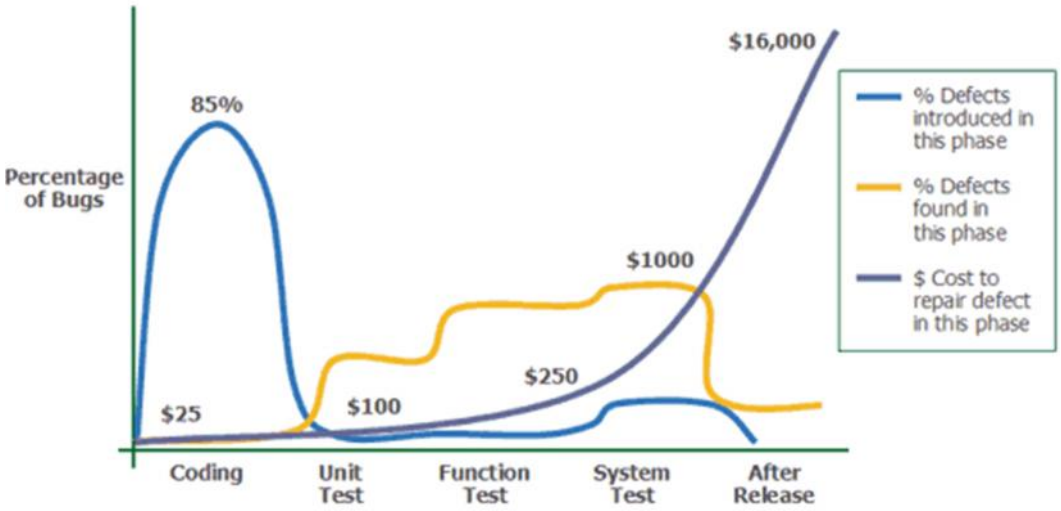

Fig. 1 Source: Applied Software Measurement, Capers Jones, 1996

With no effort put into unit- and integration testing, further agile and continuous integration techniques mentioned in D9.1 cannot be applied.
Writer of this report urges software developers to proceed with previous commitments (D9.1) towards agile and continuous software development when developing ebbits modules.

3.4 Results from Java Sonar

3.4.1 Sonar setup environment

Similar to D9.2.1 Java Sonar was used in current integration report. Details about the tool can be found in the previous integration report, chapter 3.1.5.

Analysis results were created with the Sonar version 3.1. Sonar was installed on a Debian 6.0.5 VM. Tests scripts were triggered by Sonar runner 1.3 (recommended by Sonar). For the report creation, additional plugins were installed:

- Technical Debt Plugin 1.2.1
- C# Plugin Ecosystem 1.3 (without ndeps-plugin)

At creation of the previous D9.2.1 no analysis for C# code was possible. This has changed due to the development achievements of the C# Plugin Ecosystem.

3.4.2 Java code analysis

The current section describes Sonar results of the ebbits java code base.
The following java modules were analysed:

- NetworkManager
- CryptoManager
- OntologyManager*
- TrustManager
- EbbitsContextManager*
- LinkSmartManagerConfigurator
- PWAL*
- LinkSmartMiddlewareAPI
- LinkSmartWSProvider
- LinkSmartMiddlewareClients

(*) ebbits specific code

Over 53.000 lines of total java code were scanned by Sonar. Over 50% represents ebbits specific code. There are 351 files, 97 packages, 415 classes and 3475 methods. 23.3% of total code is commented. 69% of the API is commented. 32.9% of the code, are duplications.

As stated in D9.2.1 duplications are partially the result of automatic code generation for web services. This can be omitted in future by refactoring web service stacks (stubs & skeletons) into one package. A more specific look reveals that not less than 70% of the LinkSmart duplications come from automatic web service code generation. In the ebbits specific modules 95% of the duplications come from automatic code generators. So only approximately 5% of hand written code shows symptoms of duplication. This is a very good value. Compare it with different examples: Jboss 8.1% duplications (Nemo,Jboss) , Apache 8.2% (Nemo,Apache). With such knowledge in mind , code duplications are not a real issue for the java code developed during the ebbits project.

The biggest contributor of such automatic duplications is eu.ebbits.pwal.impl.driver.pwaldriver_plc.support_libs.opc.OPCXML_DataAccessStub from the PWAL module. This module should be reviewed for possible refactoring opportunities.

72.8% of total code base complies with static analysis rules. Overall complexity of code is 3.5 per method. The total technical debt is around 26,8% . As stated in previous deliverable, duplications create big chunk of the total technical debt. Clearing the technical debt from the whole LinkSmart middleware would take estimated 914 man days.

Statistics for total LinkSmart middleware code analysis can be seen from Fig.2. As you can see from Fig.2, unit test coverage analysis is skipped due to the mentioned lack of unit tests.

For the analysis of code violations, the default “Sonar way” profile with 115 rules was chosen. Regarding the rule violations there are a few worth mentioning:

- Emtpy if Stmt
- Security – Array is stored directly
- Unconditional If Statement
- Avoid Catching Throwable
- Useless Operation On Immutable
- Equals Hash Code
- Equals Null

Developers should review them in the future. For exact definitions, please read the Sonar documentation.

When only ebbits related modules are taken into account, still 259 man days have to be invested to clear the debt (see Fig.3). Compared with the whole LinkSmart middleware, the ebbits specific modules have lower complexity, more duplications (PWAL module), slightly better documentation and rules compliance. The ebbits specific modules also lack unit tests.

The technical debt plugin masks the information about test coverage (“No information available on coverage”). Because of it the technical debt pie-chart does not show a coverage slice. Nonetheless the missing test coverage remains the biggest debt for the whole project. All other factors like duplications, rule violations, comments or complexity contribute much less to the overall debt.

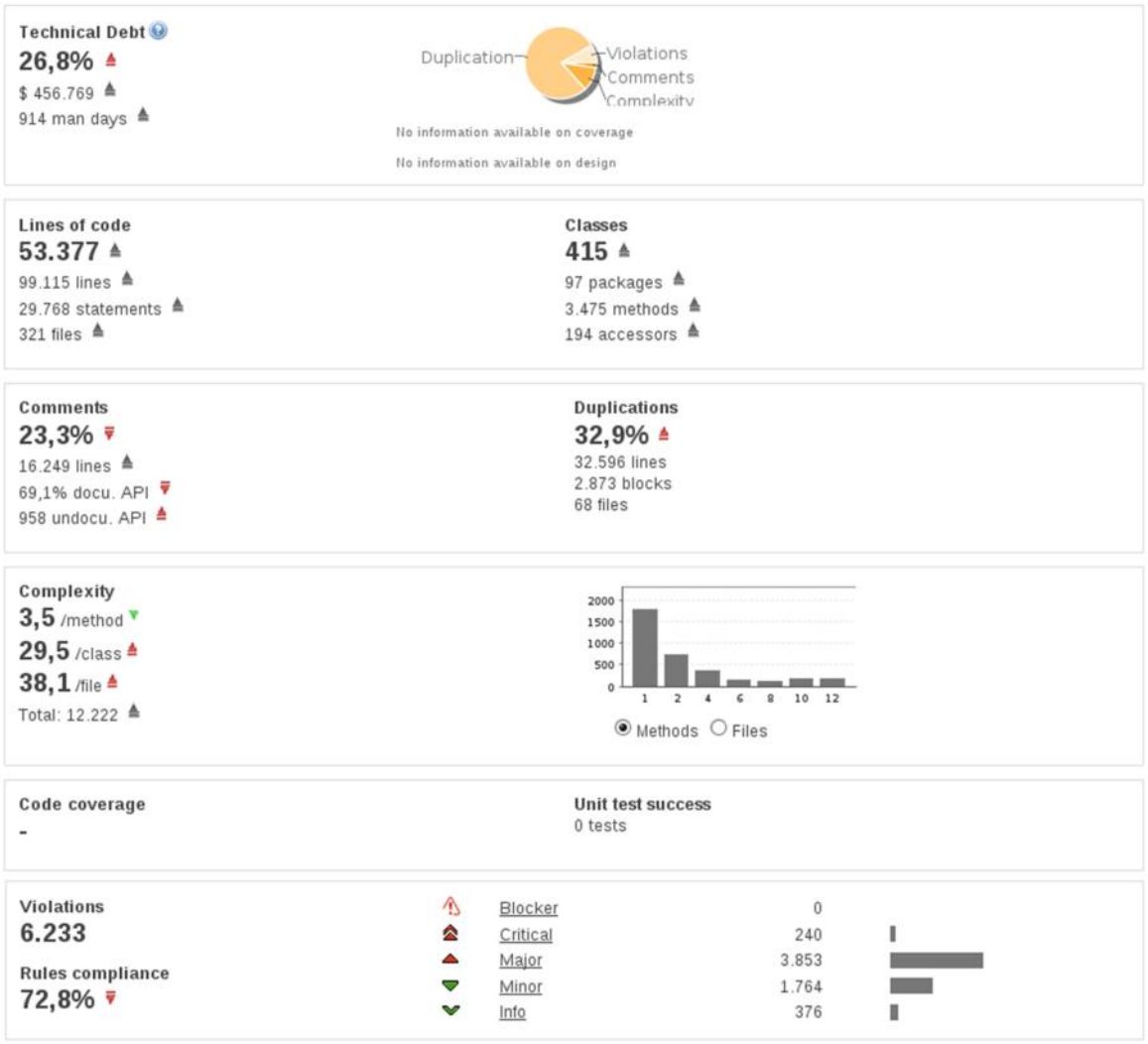

Fig. 2 Sonar results for LinkSmart total java code base

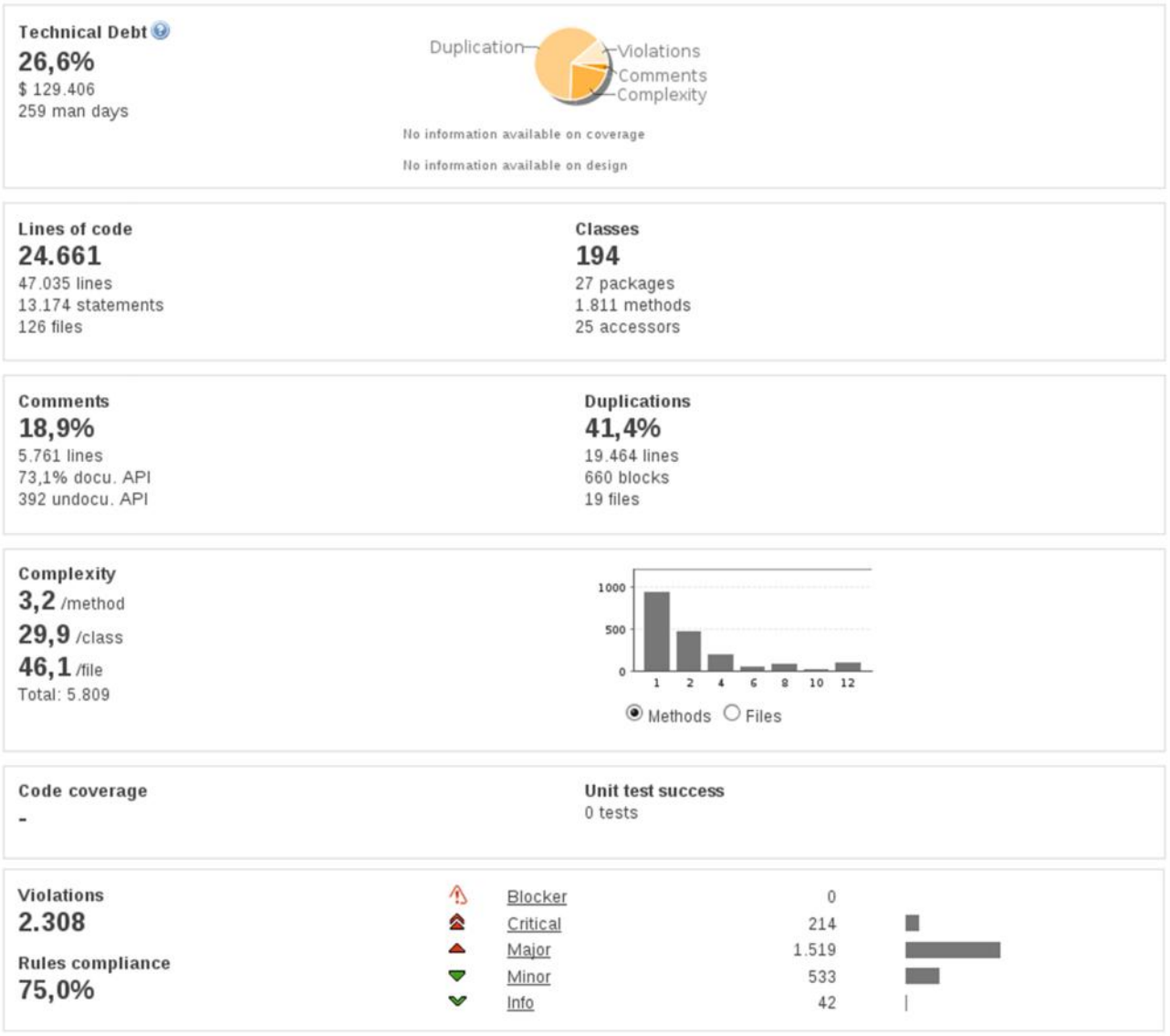

Fig. 3 Sonar results for LinkSmart Ebbits specific java code base

Compared to previous deliverable (D9.2.1), one can see a improvement.
Rules compliance enhanced from 60% to 72% in total LinkSmart code base.
The more important API documentation increased from 31% to 69%. Duplication rate stays at similar level compared to previous report. The code complexity is slightly higher in current code base. It went from 3.1 to 3.5, which makes 12% complexity increase.

Also the total number of lines was reduced from 133k to 53k which increased the total maintainability.

Statistics for ebbits specific modules are even better.
Rules compliance increased to 75% from previous 60%. API documentation rose to 73% from 31%. Duplication rate is around 5%, which is quite low. Complexity increased by only 4%.

3.4.3 C# code analysis

Parts of the ebbits code base are written in C#. Contrary to D9.2.1 results, current report is able to present Sonar based analysis of the C# code.

The ebbits repository contains three C# based modules, namely:

- EventManager
- EventProcessingAgent
- LMStationProxy

The total number of C# lines of code is around 5000. There are ~70 files, ~80 classes and 500 methods.
The total C# code is commented at rate of 14%. API comment rates are at 34% . Compared to Java code, the duplication rates are low at around 2-10% . Static rule compliance is at 45%. With 1.5 per method, the complexity of code is rather low.

The total technical debt can be estimated with 22%. It is estimated 37 man days are necessary to clean up the accumulated debt.
Also no unit tests are provided by the code base therefore test coverage analysis has to be skipped.

The following figures 4,5 and 6 depict Sonar statistics for all three C# modules. An accumulated analysis for all modules was not possible. The Sonar C# plugin requires one project file (.sln) per run, but there are three of them in the repository.

It can be seen that code coverage seems to be the biggest debt for all three modules. It is followed by rule violations and missing comments in one case (LMStationProxy).
For the analysis of code violations, the default “Sonar C# Way” profile with predefined 316 rules was chosen.

The code also contains some rule violations. Those violations should be reviewed in the next iteration. Here a few of them are listed:

- File may only contain a single namespace
- Sections of code should not be “commented out”
- Fields must be private
- Interfaces names must begin with ‘i’

In current report the C# code was analysed for the first time, thus the outcome cannot be compared with results achieved from previous reports.

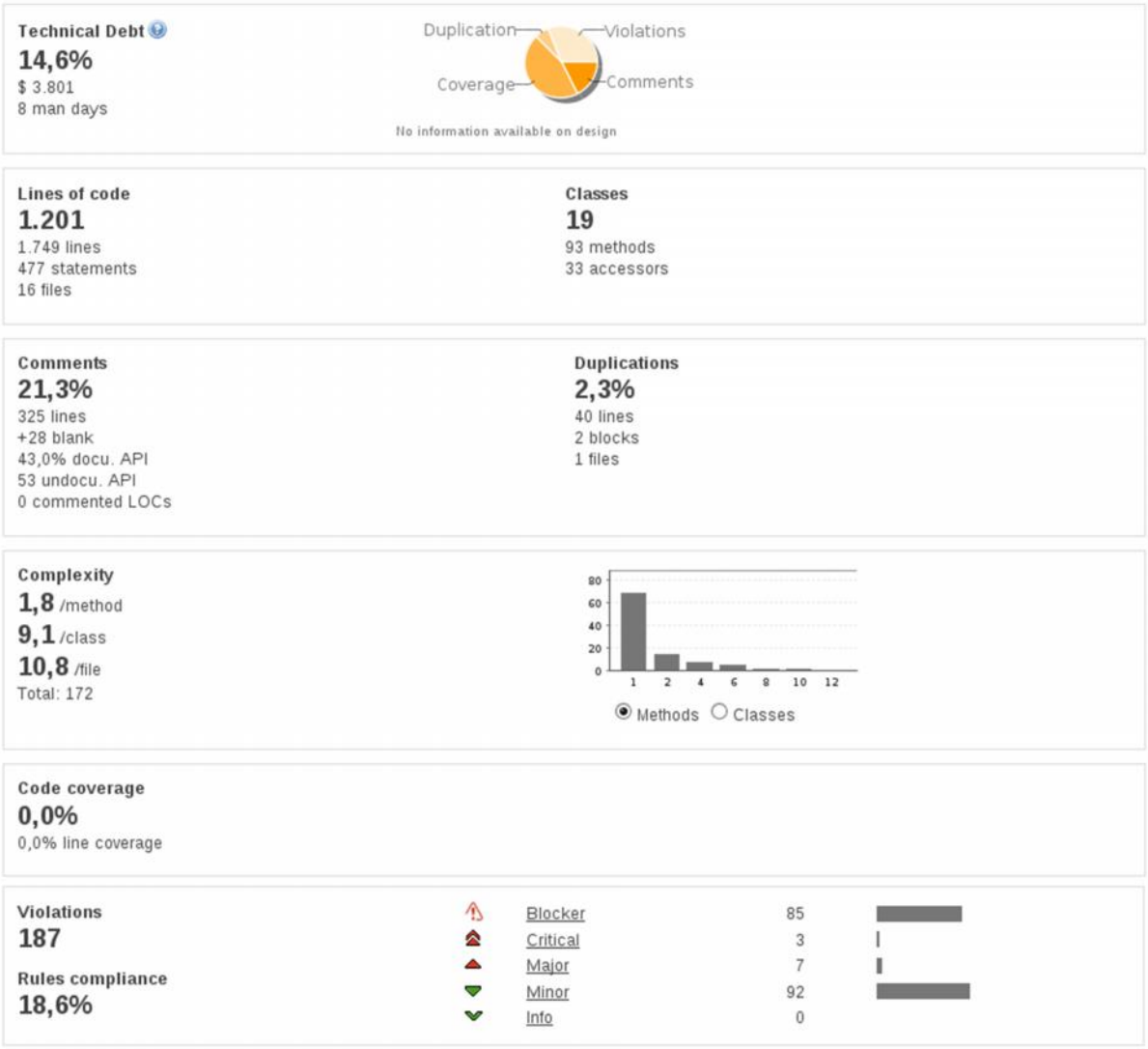

Fig. 4 Sonar results for EventManager C# module

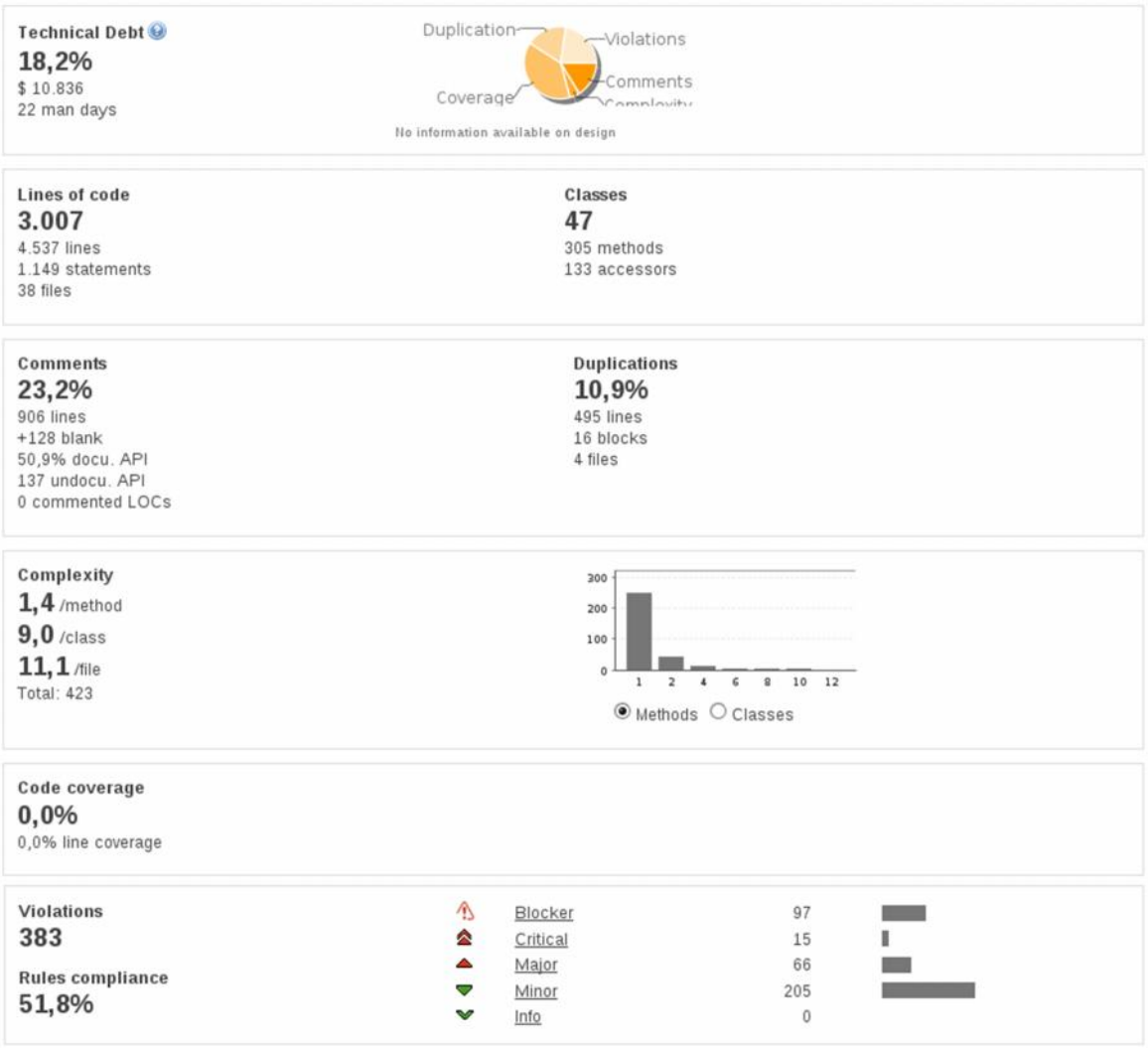

Fig. 5 Sonar results for EventProcessingAgent C# module

Fig. 6 Sonar results for LMStationProxy C# module

3.5 Test Beds

As stated in D9.1 Test and integration plan we will create specific “test beds” for verifying architecture quality criteria’s such as scalability et c. These ” test beds” will not to be fully fledged test beds in the classical sense since this would be out of the scope of the project, instead they should focus on certain quality attributes.

One test bed has been created for testing the ebbits eventing scalability and stability. The test bed consists of two main components see Figure X below:

- The Event Producer that creates events that either go to the LinkSmart Event Manager or directly to the ebbits Event Processing Agent (EPA).
- The Event Consumer that receives events from both the EPA and the Event Manager

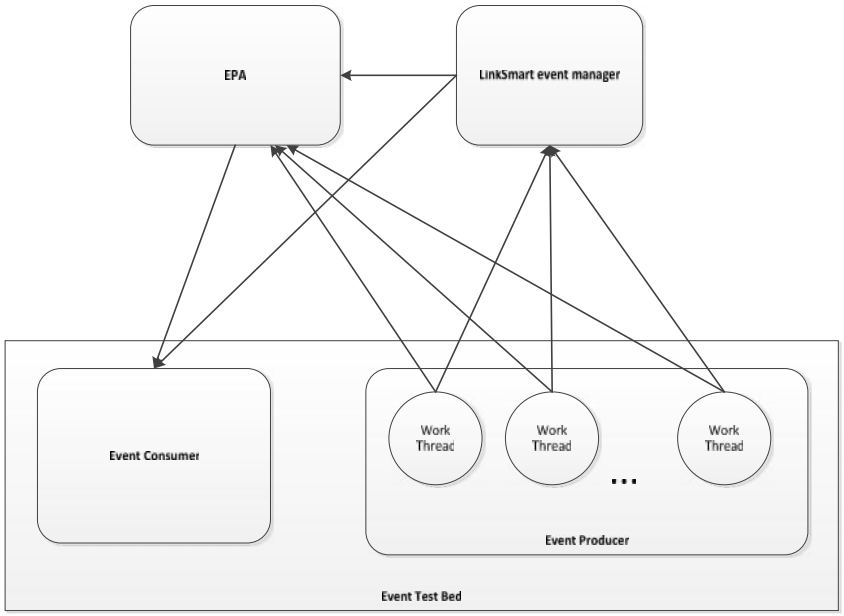

Fig. 7 Event Test bed

The main purpose of the test bed is to simulate very high event loads to be able to verify the behaviour of the ebbits eventing when it is under stress and to make sure that it does not loose events.

The basic functionality is quite simple:

1. A number of work threads that send events are started
2. At a given interval these will send a message to the LinkSmart Event Manager or to the EPA.
3. The contents of the message encode a serial # and the ID of work thread
4. The Event Consumer receives the outcome from the EPA
5. The event consumer checks the contents for the serial # and the ID to make sure that no events are missing
6. And it also measures the throughput, i.e. events per second.

The test bed is configurable with respect to the number of work threads that will be created, which events are sent.

The intention is to evolve this test bed to a more general development support tool for testing the scalability of developed rules in the EPA so that is possible for the developer to make dry runs of event loads.

4. Overview of Modules, Subsystems and Systems

As for the end of the previous iterations, the project has developed demonstrators for the manufacturing and traceability scenarios, based on a subset of the components of the complete ebbits platform.

Since ebbits is a four years iterative project, the starting point for the demonstrator development is the validation of the results achieved during the first iteration, that have been the feedback for the workpackage 2 and are driving the continuous re-engineering process of the requirements.

The activity during this second year has focused on the refinement of the modules introduced with the previous demonstrators. Each workpackage in relation to its proper sphere of responsibility contributed to this development.

The current ebbits prototype platform includes a set of common components from the two application demonstrators and a core set of middleware components being adapted from the Hydra/LinkSmart middleware. Starting from this common architecture two different demonstrators (one for each end-user environment) have been deployed.

The following sections describe more in detail the architecture developed for each demonstrator.

4.1 Manufacturing components

The main task of the manufacturing environment for the M24 demonstrator is to develop a platform able to connect the field with a mobile consumer device able to retrieve data from the devices involved in a typical automotive manufacturing process. To this aim the ebbits platform can be split into three different layers as shown in Figure 8:

- Hardware: this layer includes devices involved in the production process.
- Ebbits gateway: this level includes the PWAL (Physical World Adaptation Layers) and the proxy modules. The main aim of this layer is to incorporate all the APIs needed to interface the industrial devices with the ebbits platform and expose their functionality to the LinkSmart network.
- The ebbits Cloud: this is the upper level of the ebbits platform and it receives data from the ebbits gateway and exposes this information to the mobile devices through a web-service. Part of this layer are the Event Manager, the Ontology Manager, the Context Manager, the BRE (Business Rule Engine), the Restful WS Interface and the GUI (Graphic User Interface). The main task of this level is to manage the data coming from the gateway and to expose them to the end user through the use of Web based applications.

During the flow from the gateway to the GUI, modules like the ontology manager and context manager take part.

The ontology manager has the aim of implementing a semantic model in ebbits. The context manager interprets the data provided by the sensors

Detailed descriptions of each module can be found on specific technical deliverables such as Deliverable D8.5.1 and D9.4.

The current architecture could be probably subject to further evolution steps in the upcoming iterations in function of the lesson learned during the current iteration.

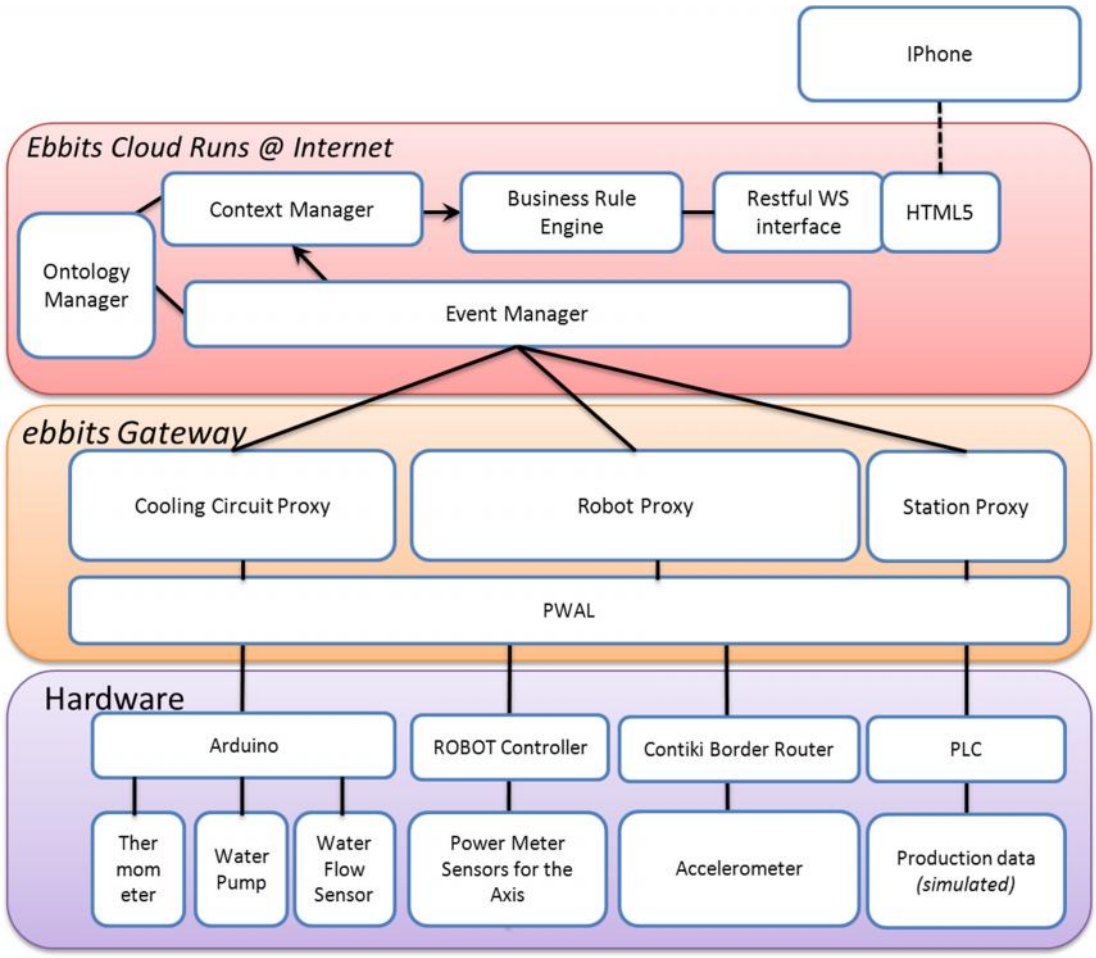

Figure 8: Manufacturing scenario component architecture (ebbits cloud

4.2 Traceability components

Figure 9 shows the parts of the traceability scenario for the final M48 demonstrator. The upcoming M24 demonstrator will not have all components available or in their complete form. The top part of the figure, that is, the external services are replaced with an online food traceability simulator which is used to input data into the ebbits core and later extracted via the traceability clients.

In the middle part are the ebbits core components and they are used as follows:

- Business Rule Engine/Service Orchestrator, selects and schedules the set of services to be invoked depending on events and rules. This will invoke the different external services to compile the information for the application.
- EPA (Event Processing Agent), receives and semantically enriches events
- Context Manager, maintains a model of the context for apps and services
- Identity Manager, provides mappings and resolution of identifiers between different id spaces.
- Event Storage Manager, maintains an event history
- Process Model, holds a description of the supported process,
- Business Rules, generic rules to support service selection and invocation, e.g., given a meat product ID, find its origin in the production chain.
- Inter-business channel services provide interfaces or links to external services.

The bottom part of Figure 9 shows the components interacting with the consumers. The smart phone app provides a graphical user interface to the information contained in the ebbits core as presented by the 'Client Traceability Services'.

The next iterations will put focus on the integration of external services in order to reduce the amount of simulated data.

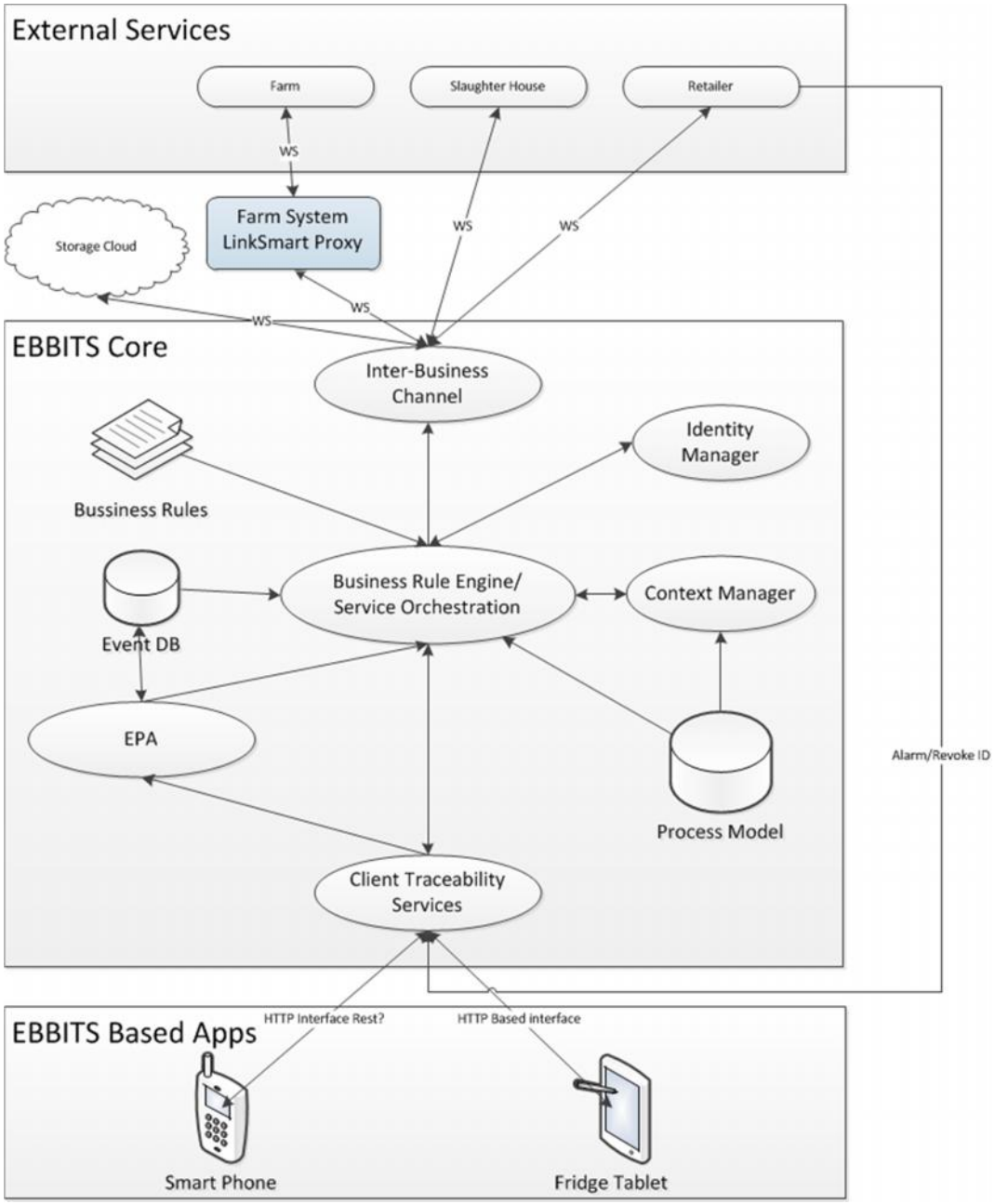

Figure 9: Traceability scenario component architecture

5. Lessons Learned

Lesson Learned LWP9 -01

As reported previously, the Java code analysis indicated a component inside the PWAL as the biggest contributor of automatic duplications. Such component is the stub automatically generated from the OPC-XML DA web service used for interfacing the PWAL with PLCs.
In order to cope with this issue and improve the quality of the software, three approaches may be pursued:

1. since OPC-XML DA is a standard, there might be already an optimized (and compiled) stub of such web service, thus avoiding the use of the automatically generated one;
2. analyse the current stub and try to optimize it, eliminating if necessary, those services not used by the PWALDriver;
3. study another approach for interfacing PLCs without the need of an OPC-XML DA server, for instance, developing a driver for the Siemens 7 protocol used for the OPC Server to communicate with Siemens PLCs.

In conclusion, this component will be reviewed and refactored according to the best strategy to follow.

Lesson Learned LWP9 -02

Even if the previously reported code analysis showed an increased availability of code documentation up to a 73% for Java code with respect to the previous year report, it is still far from a complete documentation, mostly in C# code which stuck just at a poor 34%.
Developers should commit to improve these figures and keep up to date the documentation of the components and mostly their APIs, following coding standards and best practices individuated in ebbits Quality Handbook.

Lesson Learned LWP9-03 : automatic build environment

Regarding the life cycle process of the code, a iterative integration and development environment could tackle a lot of integration problems we encounter today . The foundation of this was laid with the introduction of common SVN server. It can and should be extended in the future with e.g. automatic build server. Such agile technologies are mentioned in D9.1 but not fully introduced into the ebbits software development process.
Integration environments are becoming de-facto standard among software companies, but also EU projects. We are currently evaluating possible solutions for the java code base. When the evaluation process is over we will introduce the
integration environment to ebbits.

6. Conclusion

This document is a second in a line of deliverables reporting about integration and quality assurance progress in ebbits.
Contrary to the first report, this document skips the theoretical introduction to the topic and concentrates on the analysis of current code base. It also describes current test beds and modules.

The positive changes regarding the java code shows on-going efforts towards better integration and quality of the code base. It also shows the significance of such reports.
The previously mentioned (D9.2.1) problems like insufficient commit activity and organic growth of the repository are less severe than last year. Especially commit activity increased, compared to the data from last year.
The code duplication still remains the biggest issue for the java code base.
The main contributors of code duplication, namely the automatic stub generators are well known. Proposals for solution of this problem are included in current document.
The overall increase in quality of LS code base can be explained with partial implementation of agile methods for software development.

No #C code base comparison with previous results from D9.2.1 was possible. This option will be available in the next annual report. The #C code base is in a quite good state so no big changes are expected here. The main issue here are the low documentation rates of the code. This should be tackled during the next iteration.

The lessons learned provide helpful hints for future integration activities and – especially with regard to quality assurance methods – might be valuable to other projects as well.

7. References

(Nemo,Jboss) http://nemo.sonarsource.org/dashboard/index/176173
(Nemo,Apache) http://nemo.sonarsource.org/dashboard/index/Apache